August 1994

OS–9 International

Issue 2/94

EFFO OS9

European Forum For OS–9

8606 Greifensee, Switzerland
Fax +41 1 940 38 90
email os9int@effo.ch

sFr. 10.00
ISSN: 1019–6714

startup

Surprise, surprise, here we are again. We are sure that nobody would ever have expected another issue of OS-9 International after such a short time. So even did we.

If you take a closer look at this issue, you may notice some layout changes. The reason is easy to explain: all former issues were produced using TEX on a NeXT station, this is the first one created using a Macintosh computer. It was not just a funny idea to change the system, the background is more serious. OS-9 International originally was Marc Balmer's idea. He not only recognized that there is a demand for a publication dealing exclusively with OS-9 but he also put this idea to reality and was responsible for all issues released up to now as publisher and editor. In fact, the increasing number of inquiries about OS-9 International proves him right. Unfortunately, he decided to resign from these activities, since he no longer can afford the time needed to produce future issues of this journal.

Marc's decision has brought the European Forum For OS-9 into a peculiar situation: EFFO lost its official organ. Therefore, EFFO decided to take over the responsibility for OS-9 International. Hence, starting with this issue, please blame EFFO directly for everything that goes wrong with OS-9 International. In accordance to the decisions taken at the 1994 Annual General Meeting, there will be three issues per year.

Marc, we will do our best to continue your work.

In consequence, neither the layout nor the contents will undergo a principal change. There is really no shortage of ideas for articles. Besides articles that focus on problems primarily of interest for system programmers, we will increase the number of articles to help users coping with everyday's problems. In addition, we will present the EFFO PD collection in some greater detail and bring suggestions of how to make life with OS-9 more comfortable. The latter was always EFFO's most important goal.

We hope you will profit from OS-9 International, enjoy this new issue and stay with us in the future.

Werner Stehling

Peter Dibble at CERN

Martin Merkel

What's About to Be New

Microware is currently preparing the release of OS–9 Version 3.0. A prerelease has been distributed to the local Microware agents where final tests, mainly on packaging, are done. The second major new upgrade is Ultra C Version 1.1 which is shipping now. The near future will bring Ultra C Version 1.2, an enhanced version of FasTrak, a new release of OS–9000 as well as a C++ front end to Microware's Ultra C compiler. Mid 1994 we will see a port of OS–9000 to the PowerPC and later to the MIPS RISC architectures. Further away in the queue is Microware's multiprocessor operating system, Hydra, based on OS–9000. OS–9 Version 4.0 is currently scheduled for 1995, as well as OS–9000 Version 2.0.

FasTrak

Microware now has version 1.1 of FasTrak, its new cross development tool. FasTrak, which is currently supported on SUN 3 (Motorola 68k based) and SUN 4 (SparcStations), Hewlett Packard PA/RISC and Silicon Graphics workstations, is a replacement for the old cross development environment UniBridge. Ports to Microsoft Windows and native OS–9 systems will eventually be done as well. FasTrak is based on an X11/Motif user interface and includes a workbench, a graphical overlay to a text editor, a makefile editor, a source and assembly level debugger and a target profiling tool. Compared to UniBridge, FasTrak is running entirely on the host system. Communication with the target stations is maintained by two small daemons on the target that can be activated within the target startup file. This implementation scheme will allow future support for system state debugging of C code as the actual debugger task is running on the host. Future releases of FasTrak will be more integrated within Hewlett Packard's SoftBench, which is available from HP as a software product. SoftBench differs from FasTrak in that it supports software development distributed across multiple development stations. Integration into SoftBench will also allow the use of CASE tools, one of the items Microware customers were asking for. Other items which will be addressed in updates of FasTrak are support for emulators and the possibility of watching address locations in memory.

Ultra C Version 1.1

Microware has recently updated its new ANSI C compiler. Version 1.1 now shows a much improved compatibility with the old Kernighan & Ritchie compiler:

- In backward compatible mode, *#asm/#endasm* statements are allowed for embedded assembly code. See the following code example :

```
void myfunct (x, y)  int x; int y; {
#asm
   move.l %d0, (%a4)
   move.l %d1, (%a4
#endasm }
```

  The corresponding Ultra C syntax would be

```
void myfunct(x, y) int x; int y; {
_asm("
   move.l %d0, (%a4)
   move.l %d1, (%a4)
"); }
```

- Type checking has been relaxed in the backward compatible mode.
- Support of casts on lvalues has been improved in the backward compatible mode.

A new feature that has been introduced with Ultra C Version 1.1 is support for alignment of code and data segments. Alignment may be controlled both at assembler and object code linker level. *r68* interprets a new flag '-p':

```
-p<n>  Align orgs to <n> boundary. <n> may be 2, 4, 8 or 16.
```

Similarly *l68* has two new directives:

```
-b=<n> Align code and data segments to <n>. <n> may be 2, 4, 8 or 16.
-x=<n> Align execution offset to <n> boundary. <n> may be 2, 4, 8 or 16.
```

Inlining library functions has been improved as well. A new library *csl.l* is included in the distribution to support code inlining of calls to the C subroutine library, which previously has been provided only as a trap handler module. Similarly the floating point support package library *fpsp.l* has been announced, but this is not included in the distribution yet. Code with inlined *fpsp* calls will certainly be much bigger, as is for example the *fpsp* module with 37 KByte compared to the 6 kByte of *fpu040*. As a further enhancement of Ultra C, Microware is currently working on a C++ frontend to their compiler. This C++ implementation will come with a precoded C++ library. The current schedule foresees a beta test version for the end of Q1/94 and the final product release for end of Q2/94. Future releases of Ultra C will undergo a lot of improvements in the area of interprocedural optimization which is not fully exploited yet.

OS–9 Version 3.0

The major new product release from Microware is OS–9 Version 3.0. Currently OS–9 3.0 is shipped as a prerelease to the local distributors. The original schedule was for OS–9 3.0 to be released in April 1993, but bugs discovered in-house delayed shipping until end 1993. Nevertheless Version 3.0 is regarded as stable. As a matter of fact only minor changes were applied to the kernel since April 1993. OS–9 Version 3.0 introduces the following new features:

- **Preemptible kernel.** An important new feature in OS–9 Version 3.0 is that the kernel may be preempted by processes running with higher priority. This will lead to improved determinism for critical tasks. Kernel preemption may be optionally disabled in the init module. Preemption may also apply to file-managers with the exception of ISP and SBF.

- **Atomic OS–9.** OS–9 Version 3.0 includes as an optional replacement for the standard operating system kernel the so called Atomic kernel. This new kernel is primarily aimed at embedded systems as it will be approximately 15% smaller and will be about 15% faster than the standard kernel. The major features in which the Atomic kernel differs from the standard kernel are that with the Atomic kernel installed every process runs in super user group, there is no support for the system security module and debugging hooks in the kernel have been omitted.

- **IOMan** is a separate module. ⇒ More configurable. ⇒ Bigger. ⇒ Almost as fast.

- **Many new kernels.** There will be a separate kernel for each CPU type (68040, 68020/030, 68000, CPU32, ...). An optional buddy memory allocation scheme will be available which works slightly faster under normal conditions but much faster under worst case. On the other hand this implies that only the full memory area that has been allocated can be given back to the system, partial deallocation is not possible. All in all OS–9 Version 3.0 will ship with a total of 32 different kernels.

- **683xx systems.** OS–9 Version 3.0 will also include more support for the 683xx family of processors. Microware will provide bootable code for 68302, 68332, 68340, 68349 (which is a 68340 plus DMA and onchip cache) and soon 68360 systems.

- **SSM** may be used just to turn on copy back caching.

- Numerous small enhancements.

In the performance area a lot has been improved concerning the interrupt response time, which is nearly twice as fast as for OS–9 Version 2.4. Equally, interprocess communication, namely signals, named and unnamed pipes, alarms, data modules and sockets is on average 10-30% faster compared to Version 2.4. Additionally, a new interprocess communication mechanism in the form of binary semaphores is introduced with Version 3.0. Binary semaphores are approxi-

mately 10–15% faster than signals. They are implemented with 4 system calls *_os_sema_init*, *_os_sema_p* to reserve, *_os_sema_v* to release the semaphore and *_os_sema_term*.

OS-9 Version 4.0

OS-9 Version 4.0 will again be not system state compatible with OS-9 Version 3.0. The major item addressed in this release will be the improvement of file managers.

New Release of OS-9000

OS-9000 has been recompiled with Ultra C. This results in 10-20% faster code. Interrupt handling is now approximately 30% faster. The portability among PC compatibles has been improved as well. The new OS-9000 will include support for RAM sizes above 16 MBytes. VPC (virtual PC), which is included as default in the current distribution, will be optional in the next release to increase general performance. Equally the SSM is optional.

OS-9000 Version 2.0

OS-9000 Version 2.0 will benefit from many improvements introduced with OS-9 Version 3.0. It also will include modifications from experience gained with the Hydra project. This includes for example lightweight threads that are required for the multiprocessor OS-9000. Further changes will emerge out of the port of OS-9000 to the PowerPC architecture. OS-9000 Version 2.0 will also include modifications from POSIX. Nevertheless these modifications depend on the time-scale of the POSIX standardization process. To get an idea of the problems one can run into with the implementation of POSIX, Peter is currently implementing a subset of POSIX functions in the form of a library. Eventual public availability of this library will be announced in *comp.sys.os9*. Further enhancements Microware has in mind for OS-9000 Version 2.0 include, for example, that OS-9 binaries should be directly executable under OS-9000.

RISC Architectures

The current schedule foresees availability of the PowerPC version of OS-9000 for middle 1994. The port to the MIPS architecture currently has only second priority and will come towards the end of 1994.

Multiprocessor OS-9000

The Hydra project was originally intended as a VME based network implemented with shared memory on Motorola MVME147 CPU's. As network technology generally improved, the current design is more oriented towards a PDP (parallel distributed processing) system. As currently priority is given to the PowerPC port, the work on Hydra is moving ahead only very slowly.

Peter Dibble from Microware Systems Corporation visited CERN on November 26, 1993, to give an overview of new products and releases for OS-9 and OS-9000 currently under development within Microware.

Martin Merkel wrote this summary of Peter's presentation. He can be reached by email at <martinm@dxtemp.cern.ch>.

This article is reprinted from CERN's in-house newsletter "Online" with kind permission of the author.

How Long Do We Sleep?

Carsten Emde

The Sleep Command

The

```
sleep(seconds);
```

command is probably one of the most important and characteristic commands of a realtime operating system such as OS-9. The argument passed to the *sleep()* function is a 32-bit integer that defines how many seconds the current task is abandoned and other tasks may use the available processing power. If the argument is 0, the function still does something: the task falls into sleep of undefined length – it will only wake up, if it receives a signal. This, in fact, is the reason why the *sleep()* function is so important. It allows for interrupt driven timing which is much superior to any form of polled timing. Therefore, some people believe that serious source code intended for other purposes than just for in-house testing should avoid the *sleep()* function that passes any other argument than 0 or 1.

The Problem

If, however, the system should be suspended for a given time, what numbers may legally be used? Under OS-9 the C library transforms the *sleep()* command into the *F$Sleep* system call by converting the number of seconds into the number of system ticks that are needed to deactivate the process for the given number of seconds. This is done using the global variable *D_TckSec* (number of ticks per second) which normally is in the range between 10 and 1000 (corresponding to a system tick duration of 100 and 1 ms, respectively). As a consequence, the minimal sleep time that can be obtained using the *sleep()* function is 1 second and the maximum depends on the number of ticks per second. In a 10-ms system, for example, the *sleep()* function may theoretically be used to suspend a task for $(2^{31} - 1)$ / *D_TckSec* ≈ 248 days:

```
#define SECSPERDAY (60*60*24)
sleep(248*SECSPERDAY);
```

but this is, of course, a somewhat optimistic assumption, since the system may either have been re-booted or the CPU board reaches its MTBF in between.

Small timing intervals are certainly much more important than the above given long one. The C library for OS-9, therefore, includes another sleep function, *tsleep()*, that allows to suspend the current task for the number of given ticks. As in the above case, *tsleep(0)* suspends the system until a signal is received. When small intervals are concerned, it must also be considered that the *tsleep()* function may be executed at any time within the current time slice but the task may only restart at the beginning of a time slice. In consequence, timing accuracy is limited to the tick duration. In a 10-ms system, for example, the command

```
tsleep(1);
```

may have any duration between 0 and 10 ms and is normally used just to give up the current time slice. Similarly, the command

```
tsleep(2);
```

may have any duration between 10 and 20 ms.

More important, however, the *tsleep()* function in the above form has a significant disadvantage: it is hardware dependent, i.e. dependent on the tick duration of a particular computer system. Microware has, therefore, implemented a very useful feature: if the sign bit of *tsleep*'s argument is set, the sleep duration is not expressed in ticks but in 256th of a second. For example, the command

```
tsleep(0x80000000 + 128);
```

will suspend the current task for 500 ms.

Recommendation 1

Before using the *tsleep()* function with any other argument than 0, make sure that this is unavoidable. Often this is used for polling purposes, e.g. waiting until input arrives from standard input path:

```
main()
{
while(_gs_rdy(0) <= 0)
   tsleep(0x80000000 + 5); /* wait for 19.5 ms */
}
```

This program section may easily be replaced by a version avoiding a constant for timing:

```
#define SIG_INPUT 1000
signalhandler(signum)
int signum;
{
   ;
}

main()
{
   intercept(signalhandler);
   _ss_ssig(0, SIG_INPUT);
   tsleep(0);
}
```

The latter program section is, admittedly, somewhat longer but has the great advantage that the task will not waste any computing resources while waiting. In a practical case, the signal handler may need some improvements to appropriately treat any other arriving signal.

Recommendation 2

Whenever the

```
tsleep(ticks);
```

function is used with another argument than 0 or 1, make sure that the sign bit is set so that the sleep duration is expressed in 256th of a second. Otherwise, the software will run reliably only on a system that is set to the same tick rate as the development system, which is probably not what you want. The OS–9 NFS 1.1 package, for example, has the

```
tsleep(2);
```

command in the RPC library. When this software was started on a 1-ms system, utter chaos occurred. The NFS package could only be run on this computer after the offensive code was patched to

```
tsleep(20);
```

Unfortunately, two program versions, one for the normal 10-ms and one for the 1-ms system had to be created, since the C compiler translates the above code to:

```
7014            moveq.l   #20,d0
4E40000A        os9       F$Sleep
```

The correct code, however, requires the sign bit set and, therefore, *moveq.l* to be replaced by *move.l:*

```
203C80000005    move.l    #$80000000+5,d0
4E40000A        os9       F$Sleep
```

This code would need 4 Bytes more than the original code, so that it could not be corrected by patching.

Carsten Emde works as system integrator and software engineer for a Swiss company. He can be reached by email at <carsten@effo.ch>.

The GNU C Compilers

Stephan Paschedag

Introduction

Thanks to the Free Software Foundation (FSF) C and C++ compilers are available to the public. The C compiler was called GCC, the C++ compiler GPP (G++). Both were designed to be highly portable. Today, they are the most widely used and accepted C and C++ compilers. They have been ported to more than 20 different processors and to even more different operating systems. The author is responsible for the OS-9 ports of the current versions. This first article focuses on its history, language extensions and OS-9 specifics, and will give examples of how to call the GCC from OS-9 command line and makefiles. Forthcoming articles will describe compiler internals, libraries and other topics.

History

When Atsushi Seyama released the first port of GCC version 1.37 to OS-9 in May 1990, many OS-9 C programmers were pleased about the availability of such an efficient tool. At this time, it was mostly used to facilitate porting UNIX programs to OS-9. In addition, the optimization procedures offered by the GNU C compiler were more than welcome. Under some condition, performance could even be doubled compared to Microware's Kernighan & Ritchie (K&R) C compiler. This initial port formed the basis for further improvements and additions made by German and Swiss OS-9 programmers.

In 1991, the author ported the GPP component to OS-9. This port was facilitated by the fact that GCC was not only designed to be portable to different computer systems, but it was also designed to share its code generation part with other compilers.

In 1992, the FSF introduced version 2.0 of the two compilers GCC and GPP. This version was also ported to OS-9 by the author and first released in 1993. Due to state-of-the-art optimization procedures, version 2 provides a further performance increase of about 50% compared to version 1. The strategy used for this second port differed to some extent from the strategy used for the first port. Because of the increasing size of the GNU source package, it was no longer possible to simply use preprocessor statements. In addition, the desire to keep up with the rather quick release of new versions made such a strategy obsolete. The OS-9 specific part was, therefore, collected in separate source files.

Version 2 of the GNU compilers include even more C dialects and languages. It adds an Objective-C compiler, which implements the object oriented C dialect used on NeXT machines. There are also Numerical C and Pascal compilers available, but they have not yet been ported to OS-9 or still are in beta state.

Language Extensions

GCC is a highly optimizing C compiler that can compile source code written in C language as described by K&R as well as in ANSI C. In addition it introduces a lot of useful language extensions. Some of the more important ones are described here. It must, however, be noticed that these extensions are disabled if strict ANSI mode is specified. Certain others are disabled in the 'traditional' K&R mode. As a disadvantage, this causes trouble if GNU C extensions should be used in a general-purpose header file. The way to solve these problems is to use *alternate keywords;* they are constructed by adding '__' (double underscore) at the beginning and the end of the normal keyword.

Nested Functions

A *nested function* is a function defined inside another function (they are not supported for GPP). The name of the nested function is local to the block where it is defined. In the following example we define a nested function named *square* and call it twice:

```
double foo (double a, double b)
{
  double square (double z) { return z * z; }
  return square (a) + square (b);
}
```

The nested function can access all variables of the surrounding function being visible at the point of its definition. This is called *lexical scoping.*

Naming an Expression's Type

One can give a name to the type of an expression using a *typedef* declaration with an initializer. Here is how to define NAME as a type name for the type of EXP:

```
typedef NAME = EXP;
```

This is useful in conjunction with the feature that statements can be included in an expression. Here is how both together can be used to define a safe macro that returns the maximum of two values independently from the arithmetic type of the arguments:

```
#define max(a,b) \
  ({typedef _ta = (a), _tb = (b); \
    _ta _a = (a); _tb _b = (b); \
    _a > _b ? _a : _b; })
```

Referring to a Type with *typeof*

Another way to refer to the type of an expression is by using the keyword *typeof*. It is used similarly to *sizeof*, but the construct acts semantically as a type name defined with *typedef*.

There are two ways of writing *typeof*'s argument, with an expression or with a type. Here is an example with an expression:

```
typeof (x[0](1))
```

This example assumes that the variable x is an array of functions; the type described is that of the return values of the functions.

Here is an example with a type name used as argument to the *typeof* keyword:

```
typeof (int *)
```

The type described in this example is that of a pointer that points to an integer.

As already mentioned at the beginning, the alternate keyword *__typeof__* must be used instead of *typeof*, if the header file is intended to be used in ANSI C programs.

The *typeof*-construct can be used at any place in the source code where a *typedef* name could be used. For example, it can be used in a declaration, in a cast or inside of *sizeof* or *typeof*.

Conditionals with Omitted Operands

The middle operand in a conditional expression may be omitted. If the first operand is unequal to zero, it determines the value of the conditional expression. Therefore, the expression

```
x ? : y
```

has the value of the variable x, if x is unequal to zero; otherwise, it has the value of the variable y. This example is perfectly equivalent to the traditional syntax

```
x ? x : y
```

Double-word Integers

GNU C supports data types for integers that are twice as long as *long int*. Simply *long long int* for a double-sized signed integer can be used, or *unsigned long long int* for a double sized unsigned integer. To make an integer constant of type *long long int*, add the suffix 'LL' to the integer. To make an integer constant of type *unsigned long long int*, add the suffix 'ULL' to the integer.

Complex Numbers

GNU C supports complex data types for both integer numbers and floating numbers. The keyword *__complex__* has been introduced for this purpose.

For example, the declaration

```
__complex__ double x;
```

declares the variable x as a variable whose real and imaginary part are both of type *double*. The declaration

```
__complex__ short int y;
```

declares the variable y to have real and imaginary parts of type *short int*. The latter is not likely to be useful, but this example has been included to show that the set of complex types is complete. To write a constant with a complex data type, use the suffix 'i' or 'j'. These two suffices can be used interchangeably, since they are equivalent. For example, the statements

```
__complex__ float x = 2.5fi;
```

and

```
__complex__ int a = 3i;
```

define assignments of a complex float and a complex integer constant, respectively, to a variable of the same type.

Such a constant always has a pure imaginary value, unless a real constant is added. To extract the real part of the complex-valued expression EXP, the keyword *__real__* can be used, as in

```
__real__ EXP
```

Likewise, the *__imag__* keyword can be used to extract the imaginary part of a complex value. The operator '~' performs complex conjugation when used on a value of a complex type.

Arrays of Variable Length

Variable-length automatic arrays are allowed in GNU C. These arrays are declared like any other automatic array, but with a length that is not a constant expression. The storage is allocated at the point of declaration and deallocated when the brace level is exited. For example:

```
FILE *
concat_fopen (char *s1, char *s2, char *mode)
{
  char str[strlen (s1) + strlen (s2) + 1];
  strcpy (str, s1);
  strcat (str, s2);
  return fopen (str, mode);
}
```

Jumping or breaking out of the scope of the array name deallocates the storage. Jumping into the scope is not allowed; an error message is generated when such an attempt is made.

Macros with Variable Numbers of Arguments

In GNU C, a macro can accept a variable number of arguments like a function. In fact, the syntax for defining the macro is quite alike to that used for a function. Here is an example:

```
#define eprintf(format, args...) \
  fprintf (stderr, format , ## args)
```

In this case, the parameter *args* is a 'rest argument': it takes in zero or more arguments, as many as the parameter list of the call contains. All of them including the commas in between form the value of *args*, which is substituted into the macro body where *args* is used. Thus, the following expansion takes place:

```
eprintf ("%s:%d: ", input_file_name, line_number)
```

will be replaced by

```
fprintf (stderr, "%s:%d: ", input_file_name, line_number)
```

Note that the comma following the string constant comes from the definition of *eprintf*, whereas the last comma comes from the value of *args*.

Non-constant Initializers

The elements of an aggregate initializer for an automatic variable are not required to be constant expressions in GNU C. Here is an example of an initializer with run-time varying elements:

```
foo (float f, float g)
{
   float beat_freqs[2] = { f-g, f+g };
   ...
}
```

Labelled Elements in Initializers

Standard C requires the elements of an initializer to appear in a fixed order, the same as the order of the elements in the array or structure being initialised. In GNU C, the elements can be assigned in any order, specifying the array indices or structure field names they apply to.

To initialize a certain array element, its index value must be enclosed in square brackets and followed by an equal sign and the value. For example,

```
int a[6] = { [4] = 29, [2] = 15 };
```

is equivalent to

```
int a[6] = { 0, 0, 15, 0, 29, 0 };
```

The index values, however, must be constant expressions, even if the array being initialised is automatic.

Likewise, to initialize certain fields of a structure, the name of a field to be initialised is followed by a colon and the value to be assigned to the field in the structure initializer. For example:

```
typedef struct str {
   int a;
   double d;
   short undef;
} STR;

STR s = {a: 10, d: 1.234};
```

Case Ranges

A range of consecutive values can be specified in a single *case* label:

```
case LOW ... HIGH:
```

This has the same effect as the proper number of individual *case* labels, one for each integer value between and including LOW and HIGH. This feature is especially useful for ranges of ASCII character codes:

```
case 'A' ... 'Z':
```

An Inline Function Is As Fast As a Macro

The GCC can integrate a function's code into the code of its callers by declaring this function inline using the keyword *inline*. This makes execution faster by eliminating the function call overhead. In addition, if any of the actual argument values are constant, their known values may permit simplifications at compile time. This has the advantage that in some cases not all of the inline function's code needs to be included. The effect on code size is less predictable; object code may be larger or smaller with function inlining, depending on the particular case. Inlining of functions only makes sense if any optimization level is enabled. Otherwise, the net effect is only an increase in code size. The following is an example of an inline function:

```
inline int
inc (int *a)
{
   (*a)++;
}
```

Again, the alternate keyword *__inline__* must be used instead of *inline*, if the source code is intended to be compiled in strict ANSI C mode.

The compiler can also be directed to inline all functions being "simple enough" with the option *-finline-functions*. However, certain usages in a function definition can make it unsuitable for inline substitution.

Assembly Instructions with C Expression Operands

C expressions can be used to specify the operands of an assembly instruction, if the *asm* syntax is used. This makes it much easier to include assembly statements into C code, because the compiler takes care of providing the correct memory address or register number for any given C variable.

For example, here is how to use the *fsinx* instruction of the 68881 coprocessor:

```
asm ("fsinx %1,%0" : "=f" (result) : "f" (angle));
```

The variable *angle* is the C expression of the input operand while *result* is that of the output operand. The 'f' in front of them is an operand constraint, saying that a floating point register is required. The equal sign in '=f' indicates that the operand is an output operand. More details about operand constraints will be discussed in one of the following parts of this article.

Controlling Names Used in Assembly Code

Another feature to control the assembly code is to specify different variable names for the C and the assembly level. The keyword *asm* (or *__asm__* for strict ANSI mode) is used for this purpose. In the following examples, the C variable *foo* is renamed *myfoo*, and the function *func* is renamed *FUNC* in the assembly output, respectively.

```
int foo asm("myfoo");
```

or

```
extern func () asm ("FUNC");
func (x, y)
int x, y;
{
   ...
}
```

This can be very useful in GPP programs where the compiler normally uses a mangled name (depending on name, type and argument types) in the assembly code.

Variables in Specified Registers

A final feature mentioned here is that GCC allows to reserve specific hardware registers for a few global variables. Particularly, a register can be specified to be used for an ordinary register variable. This is exemplified in the following source section where the C compiler is forced to use always register *a4* for the variable *foo*, if possible.

```
register int *foo asm("a4");
```

The following general rules apply for specifying registers for C variables:

- Global register variables reserve registers throughout the program. This may be useful in programs such as programming language interpreters which have a couple of global variables that are accessed very often.
- Local register variables in specific registers do not reserve the registers. The compiler's data flow analysis is capable of determining where the specified registers contain live values, and where they are available for other uses.

These local register variables are sometimes convenient to be used in conjunction with the extended *asm* feature. This allows to direct the compiler to write the output of the assembly instruction directly into a particular register. However, this requires that the specified register matches the constraints defined for this output operand.

OS–9 Specific Options

To improve OS–9 support, various options have been added to the standard GCC. They are available through command line options of the compiler executive *gcc2*:

-mremote By default GCC generates 16-bit offsets to variables restricting the overall size of global variables to a maximum of 64 kByte. Programs exceeding this limit can only be linked, if this option is specified.

-mlong-calls By default GCC generates 16-bit code offsets restricting the maximum branch distance to ±32 kByte. This maximum can be extended using 32-bit offsets, if this option is specified. For execution on a 68000 target, the *-ucc* option must be specified too.

-ucc Use Ultra C utilities (opt68k, r68, l68).

-mnostack-check This option turns off the stack-checking code, which results in faster code.

-mstdstack-check GCC normally uses a faster technique for stack-checking on 68020/030/040 processors. This special technique can be turned off with this option.

-mnocom In C language, it is possible to declare the same variable in two different modules without using the *extern* prefix. To support this feature, these variables have to be mapped into the common section instead of into the standard *vsect*. All released versions of *l68* run into problems if one of these variables is initialised. Therefore, this feature can be turned off using this option.

-mgss If a C program has to be debugged on assembly level (for example drivers), it is useful to have global labels for static functions, which can be enabled with this option. The labels will be of the form <function>@<module>.

-++lib Adds the standard GPP libraries to the link list.

-uwlibs Link the original Microware *clib.l* instead of the new GNU *gclib.l*. This option cannot be used together with *-++lib*.

-ucclibs Link Ultra C libraries instead of *clib.l* or *gclib.l*.

-col The collector output file is not deleted.

-nocol Force the *collect* program to output an empty table only instead of processing all specified files. This leads to faster link times, but prevents C++ programs from working properly.

-cio Link Microware's trap library *cio.l* instead of the normal library. This option requires the *-uwlibs* option as well.

-gsrcdbg, -gg Generate a *.dbg* file for Microware's source level debugger.

-g Generate *.stb* module for debugging.

-F<prog>s<stack>p<prior>x[<xprog>]o[<opts>] Call <prog> with additional <stack>, priority <prior> and options <opts>. For example -Fl68s100p200o[-sm] will result in the command line

```
168 #100 ^200 -sm
```

The 'x' option can be used to choose an alternate *<xprog>* instead of the given *<prog>* for a cross compiler. To enclose the names for *xprog* and *opts*, '[' and ']' may not be omitted.

Examples

Normal practice of calling a compiler is not from the shell level but via the *make* program. The main advantage of this procedure is that only those files that need to be recompiled are processed. In consequence, the following examples for calling the GCC are provided as *makefiles*.

Single Source File

The first example probably represents one of the simplest ways to start the GCC from a standard OS-9 makefile. During compilation, GCC generates a message

```
gcc2: unrecognized option '-r=.' ()
```

but this is only a warning and may safely be ignored. The '-r=.' command line option is generated by the make program to advise the compiler that the relocatable output file has to be placed into the current directory, which GCC does by default.

```
#       Makefile example#1:
#       One source file
#       No debug files
#       Relocatables in source directory
#       GNU C library, no trap handler
#       No optimization
#       No additional library
#       No additional stack memory
#
TMPDIR  = /r0
#
CC      = gcc2
#
CFLAGS  = -T$(TMPDIR) -c
LFLAGS  =
#
SDIR    = .
RDIR    = .
ODIR    = /dd/CMDS
#
PROG.OBJ= prog.r
#
all: prog
        @echo "Program 'prog' made"
#
prog: $(PROG.OBJ)
        $(CC) $(LFLAGS) $(PROG.OBJ) -o $(ODIR)/$@
```

More than One Source File, Small Binary, K&R Libraries

The second example redefines the compile rule so that the above warning is avoided. This is done in the line that starts with the '.c.r:' statement. Any changes made to the built in rules can be inspected when *make* is started with the '-d' option. Furthermore, it is assumed that in this example two code modules are going to be linked into a single program. Minimal program size is ensured by using Microware's K&R libraries, the cio trap handler and enabling full optimization. The resulting code may not behave in accordance with the ANSI standard.

```
#       Makefile example #2:
#       Two source files
#       No debug files
#       Separate RELS directory
#       Microware's C library and trap handler
#       Full optimization
#       One additional library
#       No additional stack memory
#
TMPDIR  = /r0
#
CC      = gcc2
#
COMPAT  = -cio -uwlibs
OPT     = -O2 -o68 -ob -fomit-frame-pointer -mnostack-check
CFLAGS  = $(COMPAT) $(OPT) -T$(TMPDIR)
LFLAGS  = $(COMPAT) $(OPT) -L/dd/LIB -los9lib.l
#
SDIR    = .
RDIR    = RELS
ODIR    = /dd/CMDS
#
PROG.OBJ= source1.r source2.r
#
.c.r:
        $(CC) $(CFLAGS) $*.c -c -o $(RDIR)/$@
#
all: prog
        @echo "Program 'prog' made"
#
prog: $(PROG.OBJ)
        chd $(RDIR); $(CC) $(LFLAGS) $(PROG.OBJ) -o $(ODIR)/$@
```

More than One Source File, Larger Binary, ANSI Libraries

The last example also redefines the compile rule; in addition, debug files are produced for assembly and C level debugging. Optimization is disabled to facilitate debugging. The resulting binary is relatively large but the C library functions are likely to behave in accordance with the

ANSI standard. Additional stack of 4 kByte is provided as an example; its size only depends on the amount of local variables that are declared in the source files.

```
#       Makefile example #3:
#       Two source files
#       Debug files for srcdbg source level debugger
#       Separate RELS directory
#       Microware's C library, no trap handler
#       No optimization
#       No additional library
#       Four kByte additional stack memory
#
DEBUG   = -gg
#
TMPDIR  = /r0
#
CC      = gcc2
#
COMPAT  = -uwlibs
CFLAGS  = $(DEBUG) $(COMPAT) -T$(TMPDIR)
LFLAGS  = $(DEBUG) $(COMPAT) -s 4
#
SDIR    = .
RDIR    = RELS
ODIR    = /dd/CMDS
#
PROG.OBJ= source1.r source2.r
#
.c.r:
        $(CC) $(CFLAGS) $*.c -c -o $(RDIR)/$@
#
all: prog
        @echo "Program 'prog' made"
#
prog: $(PROG.OBJ)
        chd $(RDIR); $(CC) $(LFLAGS) $(PROG.OBJ) -o $(ODIR)/$@
```

The internals of GCC will be described in the next issue of OS-9 International. This article will include a description of the different passes that GCC executes to process a source file. Furthermore, the implemented optimization techniques will be explained.

The GCC and GPP compilers are available as PDs from EFFO. The current version is 2.5.8 which requires approximately 4 MByte of RAM to execute. On smaller systems with a minimum of 2 MByte of RAM, the more limited version 1.42.0 can be used. EFFO will continue to provide this smaller version but it is no longer maintained.

Stephan Paschedag works as a hardware and software engineer for a Swiss company. He can be reached by email at <stp@effo.ch>.

An sh-like Shell for OS–9

Carsten Emde

Introduction

More and more projects require the simultaneous availability of different computers such as, for example, UNIX workstations and OS–9 realtime systems. When users, however, have to work interchangeably on these systems it is highly desirable that standard programs (shell, editor, process viewer, window managers, etc.) are identical. Although most programs are available for OS–9 in the same version as for other operating systems, there still was one program lacking: the shell.

About 6 years ago, the first version of an interactive *sh*-style shell was introduced to OS–9 as part of the famous TOP project by Wolfgang Ocker, Ulli Dessauer and Reimer Mellin. Reimer wrote a large part of the initial version. About 3 years ago, when it became clear that the TOP project would not be released once more, the author of this article decided to take over the sources and to continue *sh*'s software maintenance. The *sh* for OS–9 is now available from EFFO; the current version is 1.6, edition 32.

This article shortly summarizes the program's functionality. As usual, the complete user's manual is part of the EFFO distribution.

Overview

The *sh* shell program was intended as a surrogate for the original OS–9 shell from Microware. Maximum attention was, therefore, drawn to offer the basic functionality of this shell and also of the Bourne shell in order to allow for the use of both OS–9 and UNIX shell scripts. In addition, useful features of the *csh* have been integrated. Compatibility of these three sources was, however, not always possible: the OS–9 pipe symbol ('!') is the history symbol in the *csh*, and the Bourne shell standard output append symbol ('>>') is the standard error redirection symbol of the Microware shell. In addition, command line editing using emacs commands was not always compatible to the line editing commands of the original shell: Microware shell's *redraw current line* command (Ctrl-A) sets the emacs cursor to the beginning of a line. There are, however, much more compatibilities than incompatibilities; normally, it does not take more than a day or two to become familiar with sh for OS–9.

Getting Started

The following command line arguments are available:

```
$ sh -?
Syntax:          sh [<opts>] [<scriptfile>] [<arg1>] ... [<argn>]
Function:        Operating system user interface (shell)
Options:
   -c            exit after reading and executing one command
   -d            trace commands
   -e[=<path>]   print error explanations using file <path>
   -h=<n>        set the # of history lines to <n>
   -ni           suppress shell messages
   -l            require "logout" to logout.
   -p[=<prompt>] print prompt
   -s            stop command execution
   -t            echo input lines
   -u            display references to unset variables
   -x            exit on error (default if non-interactive)
```

All options may be prefixed by an 'n' character to invert their sense.

For example

```
$ sh -ep="<$HOST/$USER/sh@>$CWD:"
```

will invoke the *sh* with the prompt

```
<thle17/carsten/sh1>/h0/SH/DOC:
```

and define the standard error message file */h0/SYS/errmsg* for displaying error texts.

Some of the options can also be set by environment variables or by built-in commands:

Option	**Environment variable**	**Built-in command**
-e=<path>	setenv EMSG_FILE <path>	
-h=<n>		history <n>
-p=<prompt>	setenv PS1 <prompt>	

In addition, the *set <option>* syntax can be used to set options within the current *sh* session.

The special setup file *.profile* is searched in the HOME directory and executed at the end of *sh*'s initialization procedure. The commands contained in the *.profile* file are executed in the same way as if they had been entered manually.

Command Line Editing and History

One of the most important features of the *sh* for OS–9 is the command line editing and history function. Editing of the current command line is always possible; the history function is only active, if enabled using the *-h=<lines>* option when *sh* is started or using the *history* built-in command at a later time.

The following editing commands are available; default keys are used, if the particular termcap entry is not specified.

Command	Default key	Termcap name
Delete previous character	Ctrl-H (BS)	bc
Delete character under cursor	Del (0x7f)	kD
Cursor backward	Ctrl-B	kl
Cursor forward	Ctrl-F	kr
Next history line	Ctrl-N	kd
Previous history line	Ctrl-P	ku
Cursor to end of line	Ctrl-Z	kH
Cursor to start of line	Ctrl-A	kh
Clear to end of line	Ctrl-K	kE
Delete current line	Ctrl-X	kL
Transpose current and previous character	Ctrl-T	kT
Toggle insert/overwrite mode	Ctrl-V	kI
Expand single file name	TAB	ta
Clear screen	Ctrl-L	kc

Editing commands and default keys

Program Execution

When a file is found in one of the *PATH* directories, an attempt is made to execute the file; default is */dd/CMDS*, if the *PATH* variable is empty. The first file found is executed, irrespective of whether it is a binary program or a procedure file. In the latter case, the file is, of course, not directly executed but forked using the shell that is taken from the first line of the file. If the procedure file starts neither with a '*' nor a '#' (forcing *sh* or *shell*, respectively), the shell program is taken from the *SHELL* environment variable. If the file is not found in one of the *PATH* directories, an attempt is made to execute it from the current execution directory, but this requires that the file has the execute attribute set. As above, also a file in the current execution directory may be either a binary program or a procedure file. The dot ('.') symbol in the PATH list (e.g. */h0/CMDS:.*) has a special meaning: It always refers to the current data directory and not

to the current execution directory, since the latter is scanned anyway. The same rule applies to the './' prefix (e.g. *./procedure*) that also refers to the current data directory.

Important Features

Regular Expressions

Regular expressions are evaluated whenever a string is entered. If, for example, all files that have 'a', 'b', or 'c' as first character are to be deleted, the command

```
del [abc]*
```

will do the job. Alternatively, the command

```
del [a-c]*
```

can be entered for this purpose.

Availability of Environment Variables

All environment variables are available as part of a command when preceded by the '$' sign. Inspecting the current setting of the *TERM* environment variable is, for example, done by entering

```
echo $TERM
```

The command

```
dir $CLIB -e
```

produces an extended directory listing of all files that are available in the linker's default library search path.

Adding the directory */h0/BIN* to the *PATH* environment variable can be done by entering

```
setenv PATH $PATH:/h0/BIN
```

Overview About Symbols, Environment Variables, Built-in Commands and Language Constructs

Most of the following symbols, variables and commands are self-explanatory and similar to other shell programs. Therefore, only short listings are presented in the following. A more detailed description can be found in the user's manual, and in the literature, e.g. [1].

Symbols

```
Pipe symbol                    '|' or '!'
Redirection symbols            '<', '>', '2>', '>>', '>+', '>-'
History event symbol           '!', if first character of an input line
Assignment symbol              '='
Variable symbol                '$'
Concurrent execution symbol    '&'
Sequential execution symbol    ';'
Evaluation symbol              '`'
Comment symbol                 '#'
Test symbol                    '['
Shell level symbol             '@'
Single quote                   '''
Double quote                   '"'
Mark symbol                    '<<'
Escape character symbol        '\'
```

Environment Variables

```
SHELL          shell program for subsequent forks
_sh            shell level
EMSG_FILE      error message file
TMPDIR         directory for temporary files
PATH           search path for programs and procedures
HOME           home directory
CWD            current working directory
LWD            last working directory
TERM           termcap settings
IFS            word separators in arguments
PS1            input prompt
PS2            input prompt, if command split up
```

Built-in Commands

```
alias <new> <old>
break
cd <dir>
chd <dir>
chx <dir>
continue
echo [-b|-n|-r] <string1> <string2> ... <stringn>
eval <cmd>
ex <cmd>
exec <cmd>
exit
export <name1> <name2> ... <namen>
false
history {<num>}
kill {-<signal>} {-s=<signal>} <pid1> <pid2> ... <pidn>
logon
logout
readonly <var>
read <var>
return {<val>}
show
true
test <opt1> <arg1> <opt2> {<arg2>}
trap <command> <number>
set <var1> <var2> ... <varn>
setenv <variable> <value>
setpr <pid> <priority>
setstack <stack>
setuid <group.user>
shift <n>
unsetenv <variable>
unset <variable>
version {<overview>}
wait <pid>
```

Language Constructs

```
for <var> {in <val1> <val2> ... <valn>}
case
esac
while <cond>
do
done
if
in
then
else
```

```
elif
until <cond>
fi
;;
||
&&
*)
```

Example *sh* Scripts

Argument Passing

The first *sh* script is intended to exemplify argument passing in general; the output lines from the *devs* program are only displayed, if they match the pattern passed as argument. If no pattern is specified, the complete program output is displayed.

```
#!sh script
if test $# -eq 0
then
  devs
else
  devs | grep $1
fi
```

Scanning Libraries for a Required Function

The second example is most useful when the linker complains about an unresolved reference, e.g. the function *myfunc()*, but there are many libraries in the standard directory that may contain the requested code.

```
for i in /dd/LIB/*.l
do
  echo Now scanning library '$i':
  rdump $i -g | grep myfunc
done
```

Swapping File Names

It may be necessary to swap the names of two files; assuming the following script is named *swap.sh*, the command

```
swap.sh file1 file2
```

will swap the names of these two files. The intermediate name will be *dummy* suffixed by *sh*'s process id.

```
rename $1 dummy.$$
rename $2 $1
rename dummy.$$ $2
```

Setting the *TERM* Environment Variable

The next example helps to correctly set the *TERM* environment variable for a given port, if other methods cannot be used. It is assumed that a Televideo 920 terminal is connected at port /t5. This script section is best placed at the end of the .login file; if started as a procedure from an *sh* session, the *setenv* command will have no effect on the current session.

```
if test $PORT = /t5
then
   setenv TERM tvi920b
fi
```

If this particular user has several terminals connected to several ports, the case statement may be more appropriate. The following example assumes three different ports, one with a Televideo and two with Qume terminals. Again, the following script section has to be placed at the end of the *.login* file; it has no effect on the current *TERM* variable, if started as a procedure.

```
case $PORT in
   /t5) setenv TERM tvi920b ;;
   /t8) setenv TERM qvt211 ;;
   /t9) setenv TERM qvt101 ;;
   *) echo "Unknown PORT: can't set TERM variable."
esac
```

References

[1] Kernighan, B. W., Pike, R., *The UNIX Programming Environment*, Prentice Hall, 1978.

Carsten Emde can be reached by email at <carsten@effo.ch>.

The sh shell program is available as PD-107 from EFFO.

_getsys();

Reto Peter

OS–9 Archive Sites

The following archive sites have been checked to carry up-to-date OS–9 software and information:

- FTP chestnut.cs.wisc.edu
- FTP lucy.ifi.unibas.ch

Monthly EFFO Meetings

Fortunately, communication via email has not yet replaced personal meetings held from time to time. Therefore, monthly EFFO meetings are organized where computer related topics in general are discussed giving specific emphasis to OS–9. These meetings are open to everybody, non-members can participate without any obligation. The greatest danger of attending the meeting is that this almost always represents the first step in becoming a regular EFFO member.

The first part of the meeting is a rather formal one based on the agenda that is distributed at least one week in advance. Main topics cover the maintenance of the EFFO PD software pool, the current state of EFFO owned computer hardware and software, and correspondence with the outside world. Later in the evening, we focus predominantly on sharing experiences and discussing the latest news regarding hardware and software in general.

The EFFO meeting always takes place at the first Friday in a month. For the rest of this year, the scheduled meeting dates are:

- Friday, September 2, 1994
- Friday, October 7, 1994
- Friday, November 4, 1994
- Friday, December 2, 1994

The official meeting starts at 8 *pm*, but most of the participants meet already at 7 *pm* to have supper together.

Looking for a central place being easy to reach, we found a nice restaurant in Brugg, a town somewhere in the middle between Zurich and Basle. We meet in a restaurant called *Rotes Haus*, which actually is a red house and can be found easily in the center of Brugg. Its address is *Hauptstrasse 7, 5200 Brugg.* If you need a map of the city you can ask for a copy via the **EFFO** address or fax.

We would like to invite you to join one of the next meetings and look forward to seeing you.

Reto Peter works as a software engineer for a Swiss company. He acts as vice president, secretary and registrar of EFFO since the early days. He can be reached by email at <reto@effo.ch>.

Imprint

OS–9 International

Published by	European Forum For OS–9 (**EFFO**)
President	Werner Stehling
Vice President	Reto Peter
Director of Finance	Stephan Paschedag
Editor-in-Chief	Carsten Emde
Design	Marc Balmer
Layout	Werner Stehling

Address
European Forum For OS–9
P.O. Box
8606 Greifensee
Switzerland

FAX +41 1 940 38 90
email os9int@effo.ch

Printed in Switzerland
ISSN: 1019–6714

Subscriptions

OS–9 International is the official organ of the European Forum For OS–9 (**EFFO**). The subscription is included with the annual **EFFO** membership fee. In addition, it is available by separate subscription for non–**EFFO** members, single issues are also available. All following prices are given in Swiss Francs, shipping included:

	Switzerland	*Europe*	*Overseas*
One year (3 issues)	25.00	30.00	35.00
Single issue	10.00	12.00	14.00

To subscribe to **OS–9 International** or to order a single issue send a letter, postcard, fax or email to **EFFO**.

Advertisements

OS–9 International is not only an ideal platform for discussing OS–9 related topics, it is also the ideal place to advertise. **OS–9 International** reaches end-users, system-software developers and, nevertheless, decision-makers.
Please contact **EFFO** for detailed information on how to place an ad in **OS–9 International**.